Java Class Loading: The Basics

August 14, 2003
By Brandon E. Taylor
Bio »
Send Email »
More Articles »

Introduction and Motivation

A practitioner's understanding of the Java platform cannot be considered complete without a solid understanding of the life cycle of a Java reference type (class or interface). The primary processes involved in this life cycle include *loading*, *linking*, *initialization*, and *unloading*. This paper discusses one of these processes, namely the loading process, in some detail.
Class loading is fundamental to two of the most compelling features of Java: *dynamic linking* and *dynamic extension*.

Dynamic linking allows types to be incrementally incorporated into a running JVM. Dynamic extension allows the decision as to which types are loaded into a JVM to be deferred until runtime. This means that an application hosted by the JVM can leverage types that were previously unknown, or perhaps did not even exist, when the application was compiled. Dynamic linking and extension support the notion of *runtime assembly of code*, a notion of critical importance to server-side containers (e.g. application servers).

A Note on Terminology

The terms "*loading*" and "*class loading*" in this article refer to the loading of reference types (classes, interfaces, and arrays). Where "*class*" or "*type*" is used, all reference types are usually implied. Explicit language will be used when this is not the case.

Type Life Cycle

Types are made available to a running program by the Java Virtual Machine. This is done through a process of loading, linking, and initialization. Loading is the process of locating the binary representation of a type and bringing it into the JVM. Linking is the process of taking the type and incorporating it into the runtime state of the JVM so that it can be executed. Initialization is the process of executing the initializers of a type (static initializers for classes; static field initializers for classes and interfaces). When a type becomes unreachable (i.e. the running application has no references to the type), it becomes eligible for garbage collection. This collection is called type unloading.
Loading a type, from a more detailed perspective, consists of three primary activities:

1. Given a type's fully qualified name, produce a stream of binary data that represents that type
2. Parse the stream of binary data produced in step #1 into internal structures in the method area of the JVM. The method area is a logical area of memory in the VM that is used to store information about loaded types.
3. Create an instance of class `java.lang.Class` that represents the type indicated in step #1

Further details about linking, initialization, and unloading are beyond the scope of this article. Interested readers are referred to the JVM Specification for more information.

Most Popular Stories

Today | This Week | All-Time

1. Using JDBC with MySQL, Getting Started
2. Creating Use Case Diagrams
3. Parsing HTML in Microsoft C#
4. An Introduction to Java Annotations

When is a Type Loaded?

This is a surprisingly tricky question to answer. This is due in large part to the significant flexibility afforded, by the JVM spec, to JVM implementations.
Loading must be performed before linking and linking must be performed before initialization. The VM spec does stipulate the timing of initialization. It strictly requires that a type be initialized on its first active use (see Appendix A for a list of what constitutes an "active use"). This means that loading (and linking) of a type MUST be performed at or before that type's first active use.

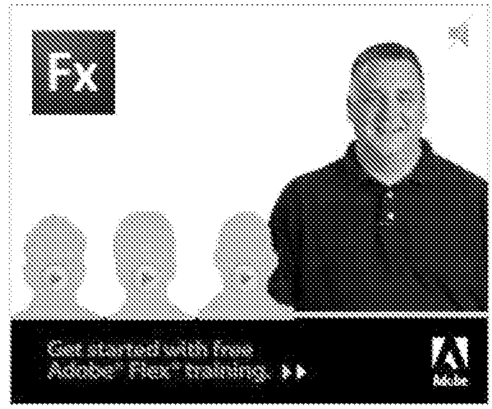
Implementations typically delay the loading of a type as long as possible. They could potentially, however, load classes much earlier. Class loaders (see below) can opt to load a type early in anticipation of eventual use. If this strategy is chosen, the class loader must not report any problem (by throwing a subclass of `java.lang.LinkageError`) encountered during loading until the type's first active use. In other words, a type must *appear* to be loaded only when needed.

Class Loaders

Classes are loaded so that their bytecodes can be processed by the execution engine of the JVM. All non-array (see Appendix B) reference types, including classes, are loaded either through the *bootstrap class loader* - sometimes referred to as the primordial class loader - or through a *user-defined class loader* - sometimes referred to as a custom class loader. The bootstrap class loader is an integral part of the JVM and is responsible for loading *trusted* classes (e.g. basic Java class library classes). User-defined class loaders, unlike the bootstrap class loader, are not intrinsic components of the JVM. They are subclasses of the `java.util.ClassLoader` class that are compiled and instantiated just like any other Java class.
Different user-defined class loaders can be specialized to provide different loading policies. For example, a class loader might cache binary representations of classes and interfaces, prefetch them based on expected usage, or load a group of related classes together (as mentioned before, however, a class loader must reflect loading errors only at points in the program where they could have arisen without prefetching or group loading).

Class loaders, in addition to loading classes, can also be used to load other types of resources (e.g. localization resource bundles) from the repositories that they manage. Further treatment of this capability is outside of the scope of this article.

1 2 »



XHTML: You can use these tags: <b> <u> <i>

(Maximum characters: 1200). You have 1200 characters left.

Please type the alphanumeric characters above and click "Submit" to continue.What's this?

I cannot read this. Please generate a

See our comment policy

Key IT Solutions

Avaya Developer Showcase

Hot List: AT&T Encouraging Standards to Assist Mobile App Developers

Hot List: Iron Speed - From Nothing to Full Web App in Five Minutes

Local Guides

- Architecture & Design
- Database
- Java
- Languages & Tools
- Microsoft & .NET
- Open Source
- Project Management
- Security
- Techniques
- Wireless

Networking Solutions

- Build Full Web Applications in 5 Minutes. Learn more.
- Whitepaper: The New Face of IT Service Management. Sponsored by IBM
- Intel Whitepaper: Client Computing with Virtual User Environment
- Intel Video: Best of Security from Manage Fusion
- Whitepaper: New Data Protection Strategies. Sponsored by IBM

WebMediaBrands

JustTechJobs

Search: Find

WebMediaBrands Corporate Info

Legal Notices, Licensing, Reprints, Permissions, Privacy Policy.

Advertise | Newsletters | Shopping | E-mail Offers | Freelance Jobs

Solutions

Whitepapers and eBooks

Oracle Whitepaper: Using Oracle In-Memory Database Cache to Accelerate the Oracle Database
Oracle Whitepapers - Innovation for IT Leaders
Microsoft Article: Make Designer/Developer Collaboration a Reality
Whitepapers: Manage Your Business Infrastructure with IBM
Whitepaper: Maximize Your Storage Investment with HP

JAVAWORLD
SOLUTIONS FOR JAVA DEVELOPERS

This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/javaqa/2003-05/01-qa-0509-jcrypt.html

Cracking Java byte-code encryption

Why Java obfuscation schemes based on byte-code encryption won't work

By Vladimir Roubtsov, JavaWorld.com, 05/09/03

May 9, 2003

Q: If I encrypt my `.class` files and use a custom classloader to load and decrypt them on the fly, will this prevent decompilation?

A: The problem of preventing Java byte-code decompilation is almost as old the language itself. Despite a range of obfuscation tools available on the market, novice Java programmers continue to think of new and clever ways to protect their intellectual property. In this **Java Q&A** installment, I dispel some myths around an idea frequently rehashed in discussion forums.

The extreme ease with which Java `.class` files can be reconstructed into Java sources that closely resemble the originals has a lot to do with Java byte-code design goals and trade-offs. Among other things, Java byte code was designed for compactness, platform independence, network mobility, and ease of analysis by byte-code interpreters and JIT (just-in-time)/HotSpot dynamic compilers. Arguably, the compiled `.class` files express the programmer's intent so clearly they could be easier to analyze than the original source code.

Several things can be done, if not to prevent decompilation completely, at least to make it more difficult. For example, as a post-compilation step you could massage the `.class` data to make the byte code either harder to read when decompiled or harder to decompile into valid Java code (or both). Techniques like performing extreme method name overloading work well for the former, and manipulating control flow to create control structures not possible to represent through Java syntax work well for the latter. The more successful commercial obfuscators use a mix of these and other techniques.

Unfortunately, both approaches must actually change the code the JVM will run, and many users are afraid (rightfully so) that this transformation may add new bugs to their applications. Furthermore, method and field renaming can cause reflection calls to stop working. Changing actual class and package names can break several other Java APIs (JNDI (Java Naming and Directory Interface), URL providers, etc.). In addition to altered names, if the association between class byte-code offsets and source line numbers is altered, recovering the original exception stack traces could become difficult.

Then there is the option of obfuscating the original Java source code. But fundamentally this causes a similar set of problems.

Encrypt, not obfuscate?

Perhaps the above has made you think, "Well, what if instead of manipulating byte code I encrypt all my classes after compilation and decrypt them on the fly inside the JVM (which can be done with a custom classloader)? Then the JVM executes my original byte code and yet there is nothing to decompile or reverse engineer, right?"

Unfortunately, you would be wrong, both in thinking that you were the first to come up with this idea and in thinking that it actually works. And the reason has nothing to do with the strength of your encryption scheme.

A simple class encoder

To illustrate this idea, I implemented a sample application and a very trivial custom classloader to run it. The application consists of two short classes:

```
public class Main
{
    public static void main (final String [] args)
    {
        System.out.println ("secret result = " + MySecretClass.mySecretAlgorithm ());
    }
} // End of class
package my.secret.code;
import java.util.Random;
public class MySecretClass
{
    /**
     * Guess what, the secret algorithm just uses a random number generator...
     */
    public static int mySecretAlgorithm ()
    {
        return (int) s_random.nextInt ();
    }
    private static final Random s_random = new Random (System.currentTimeMillis ());
} // End of class
```

My aspiration is to hide the implementation of `my.secret.code.MySecretClass` by encrypting the relevant `.class` files and decrypting them on the fly at runtime. To that effect, I use the following tool (some details omitted; you can download the full source from Resources):

```
public class EncryptedClassLoader extends URLClassLoader
{
    public static void main (final String [] args)
        throws Exception
    {
        if ("-run".equals (args [0]) && (args.length >= 3))
        {
            // Create a custom loader that will use the current loader as
            // delegation parent:
            final ClassLoader appLoader =
                new EncryptedClassLoader (EncryptedClassLoader.class.getClassLoader (),
                new File (args [1]));

            // Thread context loader must be adjusted as well:
            Thread.currentThread ().setContextClassLoader (appLoader);

            final Class app = appLoader.loadClass (args [2]);
```

```
            final Method appmain = app.getMethod ("main", new Class [] {String [].class});
            final String [] appargs = new String [args.length - 3];
            System.arraycopy (args, 3, appargs, 0, appargs.length);

            appmain.invoke (null, new Object [] {appargs});
        }
        else if ("-encrypt".equals (args [0]) && (args.length >= 3))
        {
            ... encrypt specified classes ...
        }
        else
            throw new IllegalArgumentException (USAGE);
    }

    /**
     * Overrides java.lang.ClassLoader.loadClass() to change the usual parent-child
     * delegation rules just enough to be able to "snatch" application classes
     * from under system classloader's nose.
     */
    public Class loadClass (final String name, final boolean resolve)
        throws ClassNotFoundException
    {
        if (TRACE) System.out.println ("loadClass (" + name + ", " + resolve + ")");

        Class c = null;

        // First, check if this class has already been defined by this classloader
        // instance:
        c = findLoadedClass (name);

        if (c == null)
        {
            Class parentsVersion = null;
            try
            {
                // This is slightly unorthodox: do a trial load via the
                // parent loader and note whether the parent delegated or not;
                // what this accomplishes is proper delegation for all core
                // and extension classes without my having to filter on class name:
                parentsVersion = getParent ().loadClass (name);

                if (parentsVersion.getClassLoader () != getParent ())
                    c = parentsVersion;
            }
            catch (ClassNotFoundException ignore) {}
            catch (ClassFormatError ignore) {}

            if (c == null)
            {
                try
                {
                    // OK, either 'c' was loaded by the system (not the bootstrap
                    // or extension) loader (in which case I want to ignore that
                    // definition) or the parent failed altogether; either way I
                    // attempt to define my own version:
                    c = findClass (name);
                }
                catch (ClassNotFoundException ignore)
                {
                    // If that failed, fall back on the parent's version
                    // [which could be null at this point]:
                    c = parentsVersion;
                }
            }
        }
```

```
        if (c == null)
            throw new ClassNotFoundException (name);

        if (resolve)
            resolveClass (c);

        return c;
    }

    /**
     * Overrides java.new.URLClassLoader.defineClass() to be able to call
     * crypt() before defining a class.
     */
    protected Class findClass (final String name)
        throws ClassNotFoundException
    {
        if (TRACE) System.out.println ("findClass (" + name + ")");

        // .class files are not guaranteed to be loadable as resources;
        // but if Sun's code does it, so perhaps can mine...
        final String classResource = name.replace ('.', '/') + ".class";
        final URL classURL = getResource (classResource);

        if (classURL == null)
            throw new ClassNotFoundException (name);
        else
        {
            InputStream in = null;
            try
            {
                in = classURL.openStream ();
                final byte [] classBytes = readFully (in);

                // "decrypt":
                crypt (classBytes);
                if (TRACE) System.out.println ("decrypted [" + name + "]");

                return defineClass (name, classBytes, 0, classBytes.length);
            }
            catch (IOException ioe)
            {
                throw new ClassNotFoundException (name);
            }
            finally
            {
                if (in != null) try { in.close (); } catch (Exception ignore) {}
            }
        }
    }

    /**
     * This classloader is only capable of custom loading from a single directory.
     */
    private EncryptedClassLoader (final ClassLoader parent, final File classpath)
        throws MalformedURLException
    {
        super (new URL [] {classpath.toURL ()}, parent);

        if (parent == null)
            throw new IllegalArgumentException ("EncryptedClassLoader" +
                " requires a non-null delegation parent");
    }

    /**
     * De/encrypts binary data in a given byte array. Calling the method again
```

```
     * reverses the encryption.
     */
    private static void crypt (final byte [] data)
    {
        for (int i = 8; i < data.length; ++ i) data [i] ^= 0x5A;
    }
    ... more helper methods ...

} // End of class
```

`EncryptedClassLoader` has two basic operations: encrypting a given set of classes in a given classpath directory and running a previously encrypted application. The encryption is very straightforward: it consists of basically flipping some bits of every byte in the binary class contents. (Yes, the good old XOR (exclusive OR) is almost no encryption at all, but bear with me. This is just an illustration.)

Classloading by `EncryptedClassLoader` deserves a little more attention. My implementation subclasses `java.net.URLClassLoader` and overrides both `loadClass()` and `defineClass()` to accomplish two goals. One is to bend the usual Java 2 classloader delegation rules and get a chance to load an encrypted class before the system classloader does it, and another is to invoke `crypt()` immediately before the call to `defineClass()` that otherwise happens inside `URLClassLoader.findClass()`.

After compiling everything into the `bin` directory:

```
>javac -d bin src/*.java src/my/secret/code/*.java
```

I "encrypt" both `Main` and `MySecretClass` classes:

```
>java -cp bin EncryptedClassLoader -encrypt bin Main my.secret.code.MySecretClass
encrypted [Main.class]
encrypted [my\secret\code\MySecretClass.class]
```

These two classes in `bin` have now been replaced with encrypted versions, and to run the original application, I must run the application through `EncryptedClassLoader`:

```
>java -cp bin Main
Exception in thread "main" java.lang.ClassFormatError: Main (Illegal constant pool type)
        at java.lang.ClassLoader.defineClass0(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:502)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:123)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:250)
        at java.net.URLClassLoader.access00(URLClassLoader.java:54)
        at java.net.URLClassLoader.run(URLClassLoader.java:193)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:186)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:299)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:265)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:255)
        at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:315)
>java -cp bin EncryptedClassLoader -run bin Main
decrypted [Main]
decrypted [my.secret.code.MySecretClass]
```

```
secret result = 1362768201
```

Sure enough, running any decompiler (such as Jad) on encrypted classes does not work.

Time to add a sophisticated password protection scheme, wrap this into a native executable, and charge hundreds of dollars for a "software protection solution," right? Of course not.

ClassLoader.defineClass(): The inevitable intercept point

All `ClassLoader`s have to deliver their class definitions to the JVM via one well-defined API point: the `java.lang.ClassLoader.defineClass()` method. The `ClassLoader` API has several overloads of this method, but all of them call into the `defineClass(String, byte[], int, int, ProtectionDomain)` method. It is a `final` method that calls into JVM native code after doing a few checks. It is important to understand that *no classloader can avoid calling this method if it wants to create a new `Class`.*

The `defineClass()` method is the only place where the magic of creating a `Class` object out of a flat byte array can take place. And guess what, the byte array must contain the unencrypted class definition in a well-documented format (see the class file format specification). Breaking the encryption scheme is now a simple matter of intercepting all calls to this method and decompiling all interesting classes to your heart's desire (I mention another option, JVM Profiler Interface (JVMPI), later).

Doing this interception is not hard at all. In fact, breaking my own protection scheme takes less time that it took to implement it! First, I get the source for `java.lang.ClassLoader` for my Java 2 Platform, Standard Development Kit (J2SDK) and modify `defineClass(String, byte[], int, int, ProtectionDomain)` to have some additional class logging:

```
...
    c = defineClass0(name, b, off, len, protectionDomain);

    // Intercept classes defined by the system loader and its children:
    if (isAncestor (getSystemClassLoader ().getParent ()))
    {
        // Choose your own dump location here [use an absolute pathname]:
        final File parentDir = new File ("c:/TEMP/classes/");
        File dump = new File (parentDir,
            name.replace ('.', File.separatorChar) + "[" +
            getClass ().getName () + "@" +
            Long.toHexString (System.identityHashCode (this)) + "].class");

        dump.getParentFile ().mkdirs ();

        FileOutputStream out = null;
        try
        {
            out = new FileOutputStream (dump);
            out.write (b, off, len);
        }
        catch (IOException ioe)
        {
            ioe.printStackTrace (System.out);
        }
        finally
        {
            if (out != null) try { out.close (); } catch (Exception ignore) {}
        }
    }
```

```
...
```

Note that the added lines are guarded by an `if` statement that filters for classes loaded by the system (`-classpath`) and its descendant classloaders. Also, the logging occurs only if `defineClass()` does not fail. Finally, because it is not inconceivable that more than one `ClassLoader` instance might load a class, I disambiguate the results by embedding the classloader identity in the dumped filename.

The final step is to temporarily replace `rt.jar` used by my Java Runtime Environment (JRE) (note that it could be different from the one used by J2SDK) with one that contains my doctored `java.lang.ClassLoader` implementation. Or, you could use the `-Xbootclasspath/p` option.

I run the encrypted application again and voila, I have recovered all my unencrypted and thus easily decompilable `.class` definitions. And note I have not used any knowledge of `EncryptedClassLoader` inner workings to accomplish this.

Observe that if I did not want to instrument a system class, I could have used other options such as a custom JVMPI agent that handles `JVMPI_EVENT_CLASS_LOAD_HOOK` events.

Lessons learned

I hope you found this quick excursion into details of Java classloading interesting. An important point to realize is that some tools on the market promise solutions to Java's easy reverse engineering problem through class encryption, and you should think twice before buying one. Until JVM architecture changes to, say, support class decoding inside native code, you will be better off with traditional obfuscators that perform byte-code transformations.

There is another, more useful side to such tricks as well: debugging Java classloading. Being able to get a load trace for a custom classloader could be invaluable, especially if you are trying to track down the cause of a classloader constraint violation (more on this in future **Java Q&A** posts). So, maybe Java was born to be a language for pure open source development after all? Of course, other architectures based on platform-neutral byte code (such as .Net) are equally prone to reverse engineering. I will leave you with this thought for now.

About the author

Vladimir Roubtsov has programmed in a variety of languages for more than 13 years, including Java since 1995. Currently, he develops enterprise software as a senior engineer for Trilogy in Austin, Texas.

Try Beta Log in / create account

article | discussion | edit this page | history

navigation
- Main page
- Contents
- Featured content
- Current events
- Random article

search

interaction
- About Wikipedia
- Community portal
- Recent changes
- Contact Wikipedia
- Donate to Wikipedia
- Help

toolbox
- What links here
- Related changes
- Upload file
- Special pages
- Printable version
- Permanent link
- Cite this page

languages
-
- Français

Java Classloader

From Wikipedia, the free encyclopedia

The **Java Classloader** is a part of the Java Runtime Environment that dynamically loads Java classes into the Java Virtual Machine[1]. Usually classes are only loaded on demand. The Java run time system does not need to know about files and file systems because of class loaders. Delegation is an important concept to understand when learning about class loaders.

A software library is a collection of more or less related object code. In the Java language, libraries are typically packaged in Jar files. Libraries can contain various, different sorts of objects, the most important type of object contained in a Jar file is a Java class. A class can be thought of as a named unit of code. The class loader is responsible for locating libraries, reading their contents, and loading the classes contained within the libraries. This loading is typically done "on demand", in that it does not occur until the class is actually used by the program. A class with a given name can only be loaded once by a given classloader.

Contents [hide]

Class loading process [edit]

Each Java class must be loaded by a class loader[2]. Furthermore, Java programs may make use of external libraries (that is, libraries written and provided by someone other than the author of the program) or may itself be composed, at least in part, by a number of libraries.

When the JVM is started, three class loaders are used[3] [4]:

1. Bootstrap class loader
2. Extensions class loader
3. System class loader

The bootstrap class loader loads the core Java libraries[5] (<JAVA_HOME>/lib directory). This class loader, which is part of the core JVM, is written in native code.

The extensions class loader loads the code in the extensions directories (<JAVA_HOME>/lib/ext or any other directory specified by the java.ext.dirs system property). It is implemented by the sun.misc.Launcher$ExtClassLoader class.

The system class loader loads code found on java.class.path, which maps to the system CLASSPATH variable. This is implemented by the sun.misc.Launcher$AppClassLoader class.

User-defined class loaders [edit]

By default, all user classes are loaded by the default system class loader, but it is possible to replace it by a user-defined ClassLoader (which defaults to the original root system class loader), and even to chain class loaders as desired.

This makes it possible (for example):

- to load or unload classes at runtime (for example to load libraries dynamically at runtime, even from a HTTP resource). This is an important feature for:
  - implementing scripting languages,
  - using bean builders,
  - allowing user-defined extensibility
  - allowing multiple namespaces to communicate. This is one of the foundations of CORBA / RMI protocols, for example.
- to change the way the bytecode is loaded (for example, it is possible to use encrypted Java class bytecode[6]).
- to modify the loaded bytecode (for example, for load-time weaving of aspects when using Aspect Oriented Programming).

Class Loaders in JEE [edit]

Java Platform, Enterprise Edition (JEE) application servers typically load classes from a deployed WAR or EAR archive by a tree of Classloaders, isolating the application from other applications, but sharing classes between deployed modules. So-called "servlet containers" are typically implemented in terms of multiple classloaders[2][7]

JAR hell [edit]

This section **may contain original research or unverified claims**. Please improve the article by adding references. See the talk page for details. *(August 2009)*

JAR hell is a term similar to DLL hell used to describe all the various ways in which the classloading process can end up not working.

- One case is when a developer or deployer of a Java application has accidentally made two different versions of a library available to the system. This is not considered an error by the system. Rather, the system will load classes from one or the other library. A developer who thinks he has replaced a library with a new version, but who has instead simply added the new library to the list of available libraries, may be surprised to see the application still behaving as though the old library is in use, which it may well be.
- Another version of the problem arises when two libraries (or a library and the application) require different versions of the same third library. If both versions of the third library use the same class names, there is no way to load both versions of the third library with the same classloader.
- The most complex JAR hell problems arise in circumstances that take advantage of the full complexity of the classloading system. A Java program is not required to use only a single "flat" classloader, but instead may be composed of several (or, in fact, an indefinite number of) nested, cooperating classloaders. The interactions between classloaders are too complex to be fully described here. It is sufficient to say that classes loaded by different classloaders may interact in ways which may not be fully comprehended by the developer, leading to inexplicable errors or bugs.

The OSGi Alliance specified (starting as JSR 8 in 1998) a modularity framework that solved JAR hell for current and future VM's both in ME, SE, and EE that is widely adopted. Using metadata in the JAR manifest, JAR files (called bundles) are wired on a per-package basis. Bundles can export packages, import packages and keep packages private, providing the basic constructs of modularity and versioned dependency management.

To **remedy the JAR hell** problems a Java Community Process - JSR 277 was initiated in 2005. The

resolution - Java Module System - intended to introduce a new distribution format, modules versioning scheme and a common modules repository (similar in purpose to Microsoft .NET's Global Assembly Cache). In December 2008 Sun announced that JSR 277 was put on hold [8].

See also [edit]

- Loader (computing)
- Dynamic loading
- DLL hell
- OSGi
- Apache Maven, automated software build tool with dependency management

References [edit]

1. ^ Binildas, Mcmanis (1996-01-10). "The basics of Java class loaders". JavaWorld. Retrieved 2008-01-26.
2. ^ a b Binildas, Christudas (2005-01-26). "Internals of Java Class Loading". onjava.com. Retrieved 2009-10-02.
3. ^ "Understanding Extension Class Loading". java.sun.com. 2008-02-14. Retrieved 2008-01-26.
4. ^ Dennis, Sosnoski (2003-04-29). "Classes and class loading". ibm.com. Retrieved 2008-01-26.
5. ^ These libraries are stored in Jar files called *rt.jar*, *core.jar*, *server.jar*, etc...
6. ^ Vladimir, Roubtsov (2003-09-05). "Cracking Java byte-code encryption". javaworld.com. Retrieved 2008-01-26.
7. ^ Tim, deBoer (2002-08-21). "J2EE Class Loading Demystified". ibm.com. Retrieved 2008-01-26.
8. ^ http://www.osgi.org/News/20081217

External links [edit]

- Article "Java Class Loading: The Basics" by Brandon E. Taylor
- Article "Take Control of Class Loading in Java" by Jeff Hanson
- Article "Inside Class Loaders" by Andreas Schaefer
- Article "Dynamic class loading in the Java virtual machine" by Sheng Liang and Gilad Bracha
- Article "Real-World Use For Custom ClassLoaders" by Jeremy Whitlock
- Article "Classloaders Revisited Hotdeploy" by Dr. Christoph G. Jung in *Java Specialist Newsletter*
- Article "Managing Component Dependencies Using ClassLoaders" by Don Schwarz
- Dependency manager - now at Apache incubator
- Dynamically loading classes directly from JAR files
- Free Maven Book
- ClassWorlds
- JarJar
- One-Jar
- Spring Dynamic Modules for OSGi(tm) Service Platforms
- POMStrap

Categories: Java programming language | Java platform